Using C from Python

DesertPy - Phoenix Python Meetup Group

Dave Kaspar

March 27, 2019

School of Mathematical and Statistical Sciences
Arizona State University

Outline

1. Example: Truncated search of numeric array
2. Options for calling native code
3. Overview of `ctypes`
4. Truncated search in C, `ctypes` wrapper
5. More involved usage

Consider the following task

- Given a vector $\boldsymbol{x} = (x_0, \ldots, x_{L-1})$ of length L
- Given a real value v
- Find $i = \min\{j : x_j < v\}$ or determine that no such i exists

It's trivial to implement this in Python:

```python
def firstless(x, v):
    L = len(x)
    i = 0
    while i < L and x[i] >= v:
        i += 1
    return i
```

Here we use the out-of-bounds index `i = L` to indicate no such i exists; we could also return `None`

What's wrong with this?

```python
def firstless(x, v):
    L = len(x)
    i = 0
    while i < L and x[i] >= v:
        i += 1
    return i
```

Algorithm is entirely satisfactory, but:

- Imagine this tight loop sits inside another loop, and is the hottest part of the code (not hypothetical)
- VM overhead and support of dynamic language features will dominate execution time

Built-ins less than ideal

Alternative implementation using numpy:

```
import numpy as np

def firstless_overworked(x, v):
    x = np.asarray(x)
    nz = np.nonzero(x < v)[0]
    return (nz[0] if len(nz) else len(x))
```

- The good: array operations call out to native code
- `x < v` is a Boolean array (same dimensions as `x`)
- `np.nonzero(x < v)` returns a 1-tuple holding an array of indices where `x < v` is `True`
- Algorithm is now silly, iterates through two arrays of full length `len(x)`

Truncated search example

- Was deliberately simple (but not contrived)
- Illustrates a case where writing and calling native code offers substantially improved efficiency (if this is needed)

Outline

Option: Python API

We could use Python's C API to write an extension module

- GOOD: Layer between C and Python is thin from the perspectives of:
  - Python users (can supply docstrings, function signatures, ...)
  - Execution time
- GOOD: Compile, install, distribute using distutils/setuptools
- GOOD: Mature option, documentation is pretty good
- BAD: Layer between C and Python is not thin in the sense of programmer time:
  - `PyArg_ParseTuple()`
  - Manual reference counting
  - Write algorithm in generic C, then write C wrappers for (each?) dynamic language of interest

Verdict: Suitable in certain situations but not in generality

Option: SWIG

Wrapper generator for C, C++ libraries producing bindings for *many* dynamic languages

- GOOD: Target several languages
- BAD: One size does not fit all
- BAD: Need SWIG `*.i` interface files

Verdict: May be appropriate for a substantial library, with a simple object model, when use from several dynamic languages is anticipated

Option: Boost.Python

This C++ library provides macros and functions for exposing C++ functions, classes, namespaces in Python extension modules

- GOOD: Specializing on just the pair (C++, Python) enables better matching of object models than SWIG could hope to achieve
- BAD: Limited to just C++ and Python
- BAD: The boilerplate to be written must be written in C++

Verdict: If you're using C++, this seems like the best bet

Option: Cython

```
# cyfirstless.pyx
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def firstless(double [:] x, double v):
    cdef size_t i = 0
    cdef size_t L = len(x)
    while i < L and x[i] >= v:
        i += 1
    return i
```

```
# setup.py
from setuptools import setup
from Cython.Build import cythonize

setup(ext_modules = cythonize("cyfirstless.pyx"))
```

Option: Cython

Cython transpiles to C, then compiles C to Python extension in shared object `*.(so|dylib|dll)`

- GOOD: Start writing nearly standard Python, optionally add type information
- GOOD: Interacts nicely with `numpy`
- BAD: Limited support for parallelism (essentially OpenMP for loops)
- BAD: Not portable to R, Matlab (if you care about these)

Verdict: An excellent option in many circumstances, and probably best for the truncated search example

Option: `ctypes`

The `ctypes` module from the standard library provides support for C data types (including aggregates) and calling functions in shared objects

- GOOD: No additional dependencies
- BAD: There is some boilerplate
- GOOD: There's not much of it if you're only using a few functions
- GOOD: You write the boilerplate in Python
- BAD: (Essentially) limited to C, Python

Verdict: An excellent option for one-off library calls or when you want to exercise a lot of control over the Python interface

Outline

ctypes types

- Classes for fundamental C data types, examples:
  - c_int
  - c_double
  - c_char_p (not quite fundamental, but gets special treatment)
- Classes for aggregates, derived types:
  - Structure – subclass to define your structs
  - Arrays – c_int * 16 is the *type* of an array of 16 C integers
  - Pointers – POINTER(T) is the *type* of a pointer to type T
- Each object constructed from one of these classes is backed by a memory buffer holding the equivalent C data, can pass to C functions byref()
- The values of fundamental types are accessible, settable via the .value property
- Each field of a structure is also property, gettable and settable

ctypes libraries and their functions

- Functions are available as attributes of libraries
- We specify return and argument types
- Then ctypes handles type conversions for us

```
from ctypes.util import find_library
from ctypes import * # for brevity here

libm = CDLL(find_library("m"))
libm.sqrt.restype = c_double
libm.sqrt.argtypes = (c_double,)

sqrt_5 = libm.sqrt(5)
print(type(sqrt_5), sqrt_5)
# prints <class 'float'> 2.23606797749979
```

ctypes callbacks

Most of the time we are interested in calling C from Python, but sometimes C functions take a function pointer as an argument.

```
from ctypes import * # for brevity here

# type of C function taking char * argument, returning size_t
# is CFUNCTYPE(c_size_t, c_char_p), can be used as decorator

@CFUNCTYPE(c_size_t, c_char_p)
def strlen(b):
    return len(b.decode())
```

We can pass this strlen to a C function taking an argument `size_t (*func)(char *)`.

Outline

1. Example: Truncated search of numeric array
2. Options for calling native code
3. Overview of `ctypes`
4. **Truncated search in C, `ctypes` wrapper**
5. More involved usage

Truncated search in C (obvious implementation)

```c
/* cfirstless.c */
#include <stddef.h>
#include <stdint.h>
#include "cfirstless.h"

size_t
cfirstless(const double *x, size_t len, double v)
/* Given array of length len at address x, return the first
 * index i for which x[i] < v, or return len if no such
 * index exists. */
{
  size_t i = 0;
  while (i < len && x[i] >= v)
    i += 1;
  return i;
}
```

Truncated search in C (blocked implementation)

```
typedef  double v8df __attribute__ ((vector_size (64)));
typedef int64_t v8di __attribute__ ((vector_size (64)));

size_t
cfirstless_blocked(const double *x, size_t len, double v)
/* Given array of length len at address x, return the first
 * index i for which x[i] < v, or return len if no such
 * index exists.  Process in blocks of 8 to facilitate SIMD
 * optimizations. */
{
  size_t i, j;
  size_t len_first = ((uintptr_t) x) & 63;
  if (len < len_first)
    len_first = len;
  for (i = 0; i < len_first; i++)
    if (x[i] < v)
      return i;
```

Truncated search in C (blocked implementation)

```
  v8df *xv;
  v8df vv = {v, v, v, v, v, v, v, v};
  v8di cmp;
  while (i + 8 < len)
    {
      xv = __builtin_assume_aligned(&x[i], 64);
      cmp = (*xv) < vv;
      for (j = 0; j < 8; j++)
        if (cmp[j])
          return i + j;
      i += 8;
    }

  while (i < len && x[i] >= v)
    i += 1;
  return i;
}
```

Compilation to shared library

```
gcc -O3 -march=native -fpic -Wall -shared \
  -o libcfirstless.so cfirstless.c
```

Python wrapper firstless

```python
# firstless.py
import ctypes as C
from warnings import warn
import numpy as np
import cyfirstless

try:
    lcfl = C.CDLL("./libcfirstless.so")
    for f in (lcfl.cfirstless, lcfl.cfirstless_blocked):
        f.restype = C.c_size_t
        f.argtypes = (C.POINTER(C.c_double),
                      C.c_size_t, C.c_double)
    # print("Success loading libcfirstless.so")
except:
    warn("Failed to load libcfirstless.so")
    lcfl = None
```

Python wrapper firstless

```python
def firstless(x, v, method = "C"):
    x = np.ascontiguousarray(x, dtype = float)
    method = method.upper()
    if method in ("C", "B") and lcfl:
        px = x.ctypes.data_as(C.POINTER(C.c_double))
        if method == "C":
            return lcfl.cfirstless(px, len(x), v)
        else:
            return lcfl.cfirstless_blocked(px, len(x), v)
    elif method == "CY":
        return cyfirstless.firstless(x, v)
    elif method == "N":
        nz = np.nonzero(x < v)[0]
        return (nz[0] if len(nz) else len(x))
```

Python wrapper firstless

```
    else:
        if method != "P":
            warn("firstless falling back to pure python")
        i = 0
        while i < len(x) and x[i] >= v:
            i += 1
        return i

def time():
    from timeit import repeat
    stmt = "firstless(x, v, method = '{}')"
    setup = """import numpy as np
x = np.arange(10 ** 6, -1, -1)
v = x[int(0.25 * 10 ** 6)]"""
```

Python wrapper firstless

```
desc = dict(C  = "C (obvious implementation)",
            B  = "C (blocked implementation)",
            CY = "Cython",
            N  = "Numpy (built-ins, overworking)",
            P  = "Python (purely)")
for m in desc:
    t = repeat(stmt.format(m), setup,
               number = 1000, globals = globals())
    print("Method {} 1000 times: {}".format(desc[m], t))
```

Time results (old machine)

```
In [3]: firstless.time()

Method C (obvious implementation) 1000 times:
[4.564280847000191, 4.561867651005741, 4.455790193998837]

Method C (blocked implementation) 1000 times:
[4.508035304999794, 4.651949233004416, 4.607617080000637]

Method Cython 1000 times:
[4.4173163150044275, 4.484859807002067, 4.680379589000095]

Method Numpy (built-ins, overworking) 1000 times:
[8.515583603999403, 8.813482133999906, 9.226202923993696]

Method Python (purely) 1000 times:
[139.9592333890032, 139.45108247499593, 139.2754626599999]
```

Time results (newer machine)

```
In [3]: firstless.time()

Method C (obvious implementation) 1000 times:
[1.1181618470000103, 1.1126245070190635, 1.1092812279821374]

Method C (blocked implementation) 1000 times:
[1.1583812900062185, 1.1572002400062047, 1.1530515620252118]

Method Cython 1000 times:
[1.104163747979328, 1.1027535720204469, 1.1014864780008793]

Method Numpy (built-ins, overworking) 1000 times:
[1.980167209985666, 1.981402239005547, 1.9776558729936369]

Method Python (purely) 1000 times:
[45.68740793998586, 43.17407809701399, 43.217961503978586]
```

Outline

The reason I needed `ctypes`

I've been working on a project which involves both:

- Embarrassingly parallel operations
- Shared structure

More specifically:

- Audio and symbolic segmentation using HMM ideas and dynamic programming
- Partial segmentations form a tree, good branches extended, stale branches eventually discarded

Want:

- Work in one address space
- Avoid GIL
- Drive things interactively from a REPL

A C library exposed to Python via `ctypes` achieves these objectives

`ctypes` issues arising in this work

- Who owns the memory?
  - We might care about use outside Python, so the C library does its own allocation
  - This means a few Python classes have `__del__()` methods that call `*_free()` functions in the C library
- Original object return – `ctypes` doesn't do this
  - When you extract the value of a `ctypes` object, or dereference a pointer, `ctypes` constructs a new object for the result
  - Usually not a problem if you're just reading results
- Flexible array members
  - Multiple levels of segmentation, C code aims to be generic
  - Generic `struct`s have a buffer to hold specific data, the buffer is a flexible array member
  - `ctypes` retrieves header of `struct`, from which we can get buffer size
  - Can resize buffer backing the retrieved `ctypes` `Structure`, then `memcpy` the rest of the data

Conclusion

- Need sometimes arises to write/call native code
- Several options for connecting this to Python
- With `ctypes` you define the interface in Python, suitable both for simple situations and cases where you want a lot of control
- Happy to answer any questions
- Thank you for your attention!